Programming Methodology for a General Purpose Automation Controller

M.C. Sturzenbecker, J.U. Korein, and R.H. Taylor
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

The General Purpose Automation Controller is a multi-processor architecture for automation programming. A methodology has been developed whose aim is to simplify the task of programming distributed real-time systems for users in research or manufacturing. Programs are built by configuring *function blocks* (low-level computations) into processes using data flow principles. These processes are activated through the *verb* mechanism. Verbs are divided into two classes: those which support devices, such as robot joint servos, and those which perform actions on devices, such as motion control. This programming methodology was developed in order to achieve the following goals: 1) Specifications for real-time programs which are to a high degree independent of hardware considerations such as processor, bus, and interconnect technology. 2) A "component" approach to software, so that software required to support new devices and technologies can be integrated by reconfiguring existing building blocks. 3) Resistance to error and ease of debugging. 4) A powerful command language interface.

Introduction

Recent system designs aimed at solving problems in automation control have made significant use of *multi-processing* [1-7]. Typically these systems incorporate a variable number of processors performing computations in parallel and exchanging data by means of some *interconnect technology*. These technologies are usually compatible either with a *shared memory model* [6] of data exchange or with a *message-passing model* [5,7]; occasionally, systems may exhibit features of both models. If all processors execute the same program, the system is said to be SIMD (*Single Instruction, Multiple Data*), however the greatest flexibility is achieved with a MIMD (*Multiple Instruction, Multiple Data*) system.

Multi-processor systems not only offer the prospect of increased computing power to meet the ever-increasing requirements of real-time control, they carry the potential for a high degree of configurability. One of the obstacles to rapid progress in robotics research and the deployment of programmable automation in scientific and manufacturing applications is the lack of configurability inherent in most currently available systems. The need for configurability arises from the need to integrate new devices, employ new strategies, or add processing power incrementally without making major changes to the system; for software this implies a need for "fast prototyping", the ability to construct software that can rapidly adjust to changing requirements [8,9]. However, multiprocessor systems cannot be used to tackle this problem unless another problem is simultaneously addressed: the lack of tools and methodologies to simplify the task of programming such systems.

What is a Programming Methodology?

A true programming methodology is not simply a collection of tools or techniques, rather it provides a set of concepts, usually formally or semi-formally defined, which serve as a basis for problem decomposition and program design. [10] This is a fairly definitive requirement which is not covered by vague terms indicative of a general approach, such as "top-down design" or "object-oriented programming".

A methodology has a number of advantages over a "toolbox", of which the following are representative:

- The evolution of tools and techniques does not always promote ease of use, and of themselves tools do not help to guarantee good program design. [11]
- A methodology provides a powerful language for describing the function of a program or specifying its design.
- A methodology can provide a framework for reasoning about the correctness of a program.
- Programs developed according to a methodology are often easier to maintain.

This paper describes such a programming methodology developed in conjunction with a General-Purpose Automation Controller (GPAC) project at IBM [8,9]. The ultimate aim of this methodology is to simplify the task of programming real-time, distributed systems for manufacturing engineers, control specialists, and other system users whose primary area of expertise is not computer science.

Currently, most distributed programs are written either in a concurrent language for which a distributed environment has been built (e.g., Ada [12,13]), a conventional sequential language with enhancements for distributed programming [4,7], or a special-purpose language [14]. None of these approaches is particularly adept at dealing with highly reconfigurable systems. To change the behavior of a system one must make changes to the program's source language, re-compile, re-link, re-download, etc. The usual approach is to provide primitives [15,16] (either as part of the language syntax or as system calls) for communication, synchronization, mutual exclusion, and even control over the granularity of concurrency. Application programming in such a system involves not only getting the sequential algorithms right but also managing the interaction between concurrent modules and using the primitives correctly. Many of the programs developed using these approaches are dependent for their performance, if not for their correctness, on a particular model of data exchange. Finally, there is no true methodology associated with these approaches, although useful tools such as debuggers, simulators, and syntax-directed editors are often provided.

Basic Concepts

The GPAC programming methodology depends upon a set of basic concepts relating to both hardware and software.

Hardware

The fundamental hardware concept in GPAC is that of a *real-time processor*. Each real-time processor is a distinct entity which can perform one or more computational tasks. Each real-time processor has its own instruction stream, hence the GPAC model is MIMD. A set of real-time processors sharing a common communications bus or network is a *real-time system* (RTS). Programs are developed on a *programming system* (PS) and are then downloaded into the real-time processors. Associated with each real-time processor are one or more *interrupt sources*. An interrupt source, in the abstract, is simply a request to a real-time processor to perform some work. Interrupt sources have *priorities*; work performed in connection with one interrupt source can be preempted by an interrupt source with a higher priority. Interrupt sources can be generated by other real-time processors or by devices connected to the real-time system. It is important to remember that the notions of interrupt source and priority are fundamentally abstract, and may be realized in the actual system in a number of ways. One last hardware concept is the *physical device*. A physical device is a gateway by which data can be passed to and from external hardware and is accessible by a particular real-time processor. Sensors, actuators, pendant interfaces - to name but a few - are examples of physical devices.

In GPAC, as in other systems [16], implementation of the PS is decoupled from that of the RTS, so that the architecture of the PS can be quite different from that of the RTS, as it should be since the requirements are different.

Function Blocks

The fundamental software concept in GPAC is that of a *function block* [9]. A function block is a basic computational unit assigned to one or more real-time processors; it communicates through *input ports* and *output ports*. In addition, a function block may communicate with *physical devices*, and may report *conditions*, or events which require exceptional action by the system. Finally, a function block may have some *formal parameters*, which are for its internal use only and normally not visible from outside. These five components comprise the *interface* to the function block.

A function block can be viewed externally as a "black box" which takes inputs, performs some computation, and produces outputs (and in some cases, reports conditions). We denote the inputs, outputs, devices, conditions, and parameters of a function block F by I_F, O_F, D_F, C_F, and P_F respectively.

The most basic form of function block is called an *application subroutine*. In the current system, this is coded in the C language [17] and corresponds to a C function. A set of macros is used to specify the interface; this hides any implementation details from the programmer.

An application subroutine is written in sequential code, and of itself contains no notion of concurrency. If certain coding conventions are followed, the application subroutine is also *re-entrant*, and multiple *instances* of it may be active either on the same real-time processor or on different real-time processors. A *function block instance* is obtained by *binding* the ports to specific data objects, supplying actual references for the physical devices, and values for the conditions and formal parameters. Other preconditions for the creation of a function block instance are its assignment to a particular real-time processor and interrupt source, and determination of the means by which it is scheduled.

Data Flow Graphs

More complex function blocks can be built up from simpler ones such as application subroutines. These complex function blocks are called *data flow graphs*.

A data flow graph G consists of:

- A set f_G of function blocks.
- A set I_G of input ports, a set O_G of output ports, and a set L_G of *local cells*.
- A mapping $\rho : I_G \cup O_G \cup L_G \rightarrow 2^{data_G}$,[1] where $data_G$ are the *data nodes* of G:

$$data_G = \bigcup_{F \in f_G} I_F \cup O_F$$

All the data nodes must be mapped to by ρ, i. e., for any $d \in data_G$ there exists p such that $d \in \rho(p)$.

A data flow graph is composed of communicating function blocks, but externally it is indistinguishable from an application subroutine. Data flow graphs are useful for expressing *distributed execution* of an algorithm. For example, a servo algorithm usually involves reading some value from a sensor, performing some computation, and writing another value to an actuator. In a distributed system, however, there is no guarantee that the sensor and actuator will both be accessible from the same real-time processor. Thus, in general, this algorithm cannot be executed by a single application subroutine, an instance of which is constrained to run on a single processor.

Data flow graphs are also useful for pasting together application subroutines of general utility. Perhaps we wish to add some digital filtering to the input in our servo example. This can be done most conveniently by building a data flow graph, provided some digital filtering module has already been installed as part of the software component data base.

Ports of Function blocks

Each input and output port of a function block has a *type* and a *mode*. The type is simply a C type declaration. Ports can be bound to data objects only if those objects have a compatible type. The mode of a function block port describes the relationship between the modification, or *updating* of the object to which the port is bound, and the frequency with which the function block is executed.

There are three port modes:

- *synchronous* - The object bound to the port is updated on every invocation.
- *ratio* - The object bound to the port is updated at a specified sub-frequency of the frequency of invocation.
- *asynchronous* - There is no fixed relationship between the updating of the object to which the port is bound and the frequency of invocation.

Thus, if a function block has a synchronous input port, an instance of it can be scheduled for execution only when a new value of the object bound to the port is made available by another function block instance which writes the object through an output port.

If a function block has no synchronous inputs, then it can be scheduled by assignment of an *execution interval* or a *trigger*. The execution interval specifies a frequency at which the function block must be executed; a trigger associates execution of a function block directly to occurrence of an interrupt source.

Another rule enforced by the GPAC system is that values produced by a function block are not made available to other function blocks until the former has completed its current invocation. This gives an atomic flavor to function blocks and helps to insure that the world will always be seen in a consistent state.[2]

Advantages of the Function Block Concept

Specification of real-time computations using the function block concept has the following advantages:

- Real-time requirements are separated from the executable code. This means that instances of the same code can be in-

---

[1] 2^S denotes the set of all subsets of S

[2] Errors occurring in the transmission of values over a network may, however, yield such inconsistencies; the methodology assumes that handling of these errors is transparent to the application.

voked with differing real-time requirements, that requirements can be changed dynamically.

- Function blocks have no knowledge, and consequently no dependence, on the particular data exchange model. Although the current GPAC architecture is based on shared memory, the system could be implemented using message passing without affecting the design of applications written for it.
- By standardizing the interface to function blocks and hiding the implementation details, we hope to encourage programmers to design routines that will be easily reusable. One problem which has emerged historically in the "component" approach to software design is that, in the absence of a methodology, the success of the approach is dependent upon the good judgment and foresight of programmers.
- Consistency in data typing is automatically checked. Primitives for synchronization and mutual exclusion are unnecessary, hence their misuse is impossible. The result is a system that is more robust and easier to debug and maintain.

Processes in GPAC

Work done by a real-time system is normally divided into *processes*. In GPAC, these processes are merely sets of communicating function block instances. Function block instances are *installed* before they can be executed; this consists of setting up the port bindings and attaching the function block instance to the proper processor and interrupt source. Consistent use of data objects bound to ports and of function block scheduling is checked during installation.

A typical GPAC process will consist of three phases:

- Installation of function block instances to be used.
- Execution (either once, or repetitively) of one or more function block instances.
- Waiting for termination of all the function block instances, or for a function block instance to report a condition that results in process termination.

As an example, consider a process to implement coordinated motion of several robot joints. This process requires two function blocks: one which computes the coefficients of a *trajectory* (i. e., desired joint position as a function of time) based on the current positions, target positions, speed limits, etc., and another which generates intermediate points along the trajectory and outputs these commanded positions to the joint servo modules. (The servo modules themselves are considered part of a different process, as we shall see shortly.) The first function block, called the *trajectory planner*, is executed once, the second, called the *setpoint generator*, is executed repetitively at some time interval (e. g., 20 msec). The whole process terminates when the current (sensed) position of the joints becomes close enough to the target position.

Command Lists

We now describe the above process behavior using GPAC terminology:

1. Install the trajectory planner and the setpoint generator.
2. Execute the trajectory planner once.
3. Activate the setpoint generator for repetitive execution.
4. Wait until the setpoint generator reports that the motion has completed (i. e., the sensed position is close enough), or that some other condition (unexpected force sensed, time limit exceeded, etc.) has occurred.

This description of a process is called a *command list* in GPAC. The elements of a command list are called *function block commands* and take function block instances as arguments. A process consists of a command list and a set of *termination conditions*.

Command List Transitions

In the case of robot motion, however, termination conditions are often simply signals that the system should proceed with the motion using a somewhat different plan. This is particularly true in *compliant motion* and specialized combinations of motions such as *centering grasp* [8]. In these cases the termination of a process results in a *transition* to another process.

Verbs

We now have developed the necessary concepts to define a *verb* in GPAC. A verb V consists of:

- A set f_V of function blocks.
- A set p_V of parameters.
- A set o_V of output values.
- A subset $\mathbf{inst}_V$ of p_V and a mapping $\chi : \mathbf{inst}_V \rightarrow 2^{\mathbf{data}_V}$ where $\mathbf{data}_V$ is the set of all *data nodes* in V, i. e., the union of all inputs, outputs, and physical devices in the function blocks of V:

$$\mathbf{data}_V = \bigcup_{F \in f_V} I_F \cup O_F \cup D_F$$

- A mapping $\psi : p_V - \mathbf{inst}_V \rightarrow 2^{\mathbf{vals}_V}$ where $\mathbf{vals}_V$ is the set of all *values* accessible within V, which includes all the ports as well as the conditions and parameters:

$$\mathbf{vals}_V = \bigcup_{F \in f_V} I_F \cup O_F \cup C_F \cup P_F$$

- A mapping $\omega : o_V \rightarrow \mathbf{vals}_V$.
- A set c_V of command lists, of which one is distinguished as *initial*.
- A set t_V of termination conditions.
- A mapping $\tau : \mathbf{trans}_V \rightarrow c_V$ where $\mathbf{trans}_V$ is the *transitions* of V, a subset of $c_V \times t_V$.
- A mapping $\upsilon : \mathbf{term}_V \rightarrow 2^{o_V}$, where $\mathbf{term}_V$ is the set of *terminations* of V:

$$\mathbf{term}_V = \{ t : \exists c \in c_V \ni (c,t) \notin \mathbf{trans}_V \}$$

$\mathbf{inst}_V$ are the *instantiation parameters* of the verb, and χ is the *instantiation mapping*. These elements describe the communication of data to and from the instantiated function blocks of the verb. The verb parameters not in $\mathbf{inst}_V$ are called *input parameters* and the mapping ψ is the *input mapping*. These elements describe how initial values are set up to be accessed by the instantiated function blocks of the verb. The mapping ω is the *output value mapping* which defines a set of values that may be returned from the verb.

Instantiation parameters are either data objects which can be bound to function block ports or references to physical devices. Input parameters are simply values which can be stored in the objects bound to ports or in conditions or parameters of a function block. As in the case with the mapping ρ for data flow graphs, every member of $\mathbf{data}_V$ must be a member of some set in the range of χ; that is, every data node is mapped to by some instantiation parameter.

If $\tau(c_1,t) = c_2$ then if condition t occurs while the process described by c_1 is executing, that process is aborted and the process described by c_2 commences. If no transition is specified for a given termination condition and process, then receipt of that condition within the process causes termination of the entire verb.

The mapping υ describes the *termination actions* of a verb. Each termination of a verb can return a subset of the output values defined for the verb.

Verb Instances

The instantiation parameters of a verb provide bindings for the ports of the constituent function blocks of the verb. Thus, *application* of parameters to a verb yields function block instances for each of the constituent function blocks, and the collection of these defines a *verb instance*. A verb instance in GPAC closely resembles the concept of a *task* in more traditional systems. Verb instances can be started, halted, suspended, and resumed.

The distinction between verb and verb instance can easily be appreciated by analogy with an operating system utility residing as a binary image on secondary storage. When the utility is invoked by a user, its inputs and outputs are bound to actual files and devices and a task, or main memory image, is created. Furthermore, multiple instances of the same utility can sometimes be active in the system simultaneously. The same is true for multiple instances of a verb in GPAC.

Verb instance commands are passed from the Programming System to the Real-Time System, where they are executed by the *supervisory software*. In every RTS, there is one real-time processor which is distinguished as the *supervisor*. The supervisory software is downloaded (possibly along with applications code) into this processor, and it maintains communication with the PS while the RTS is running its applications. While the PS sends verb instance commands to the RTS, the RTS sends notification of *verb instance termination* to the PS.

All real-time processors, whether or not they are supervisors, contain a *real-time kernel* which is primarily responsible for handling interrupts, context switching, and dispatching of application subroutines. The supervisor handles all activity related to:

- Installation, activation, and deactivation of function block instances.
- Transitions between command lists.
- Termination of verb instances.

State Vector Variables and Logical Devices

We have previously referred to the representation of data within GPAC without providing any details. Data which must be communicated between function blocks is represented by the concept of a *state vector variable* (SVV). It is perhaps easiest to think of an SVV as shared memory, but it need not be implemented that way. A state vector variable contains a buffer for the current value, and optionally a buffer for a set of *previous values*, in case a history needs to be maintained. State vector variables are bound to function block ports in order to effect communication between funcion block instances. Associated with each SVV is a *type*, and it can only be bound to ports of the same type.

Although a state vector variable is essentially an independent concept, the primary method of creating and using the SVV in GPAC is through the more powerful concepts of a *logical device* and a *logical device type*. A logical device type consists of:

- An optional verb.
- A specification for a set of SVVs.
- A specification for a set of parameters.

A logical device can be considered an *instance* of a logical device type and consists of:

- An optional verb instance.
- A set of SVVs.
- A set of parameters.

A robot joint is an excellent example of a logical device type. With each joint in the system we normally want to maintain the following data:

- The current *sensed* position of the joint.
- The current *commanded* position of the joint.
- The *goal*, or target position.

This data will be stored in SVVs, since it will be updated by function block instances in real time. We may also associate certain parameters with a robot joint:

- The maximum speed at which the joint can be moved.
- The maximum force or acceleration to which it may be subjected, etc.

These values do not normally vary in real time and hence do not need to be stored in state vector variables.

Logical Devices as Verb Parameters

A verb may be *applied* to one or more logical devices; alternatively, we might say that logical devices can appear as the *objects* of verbs. In this case, the state vector variables and parameters which comprise the logical device will be entered into the parameter list from which the verb instance will be constructed. Thus they can be bound to the ports and parameters of function blocks which will perform the computations associated with the verb.

A robot joint is *controlled* by some real-time process (servo). This is implemented as a verb in GPAC. The definition of a logical device type for the robot joint contains specifications for the servoing verb, the SVVs, and the parameters. Then, when an instance of a joint is created, the actual verb instance, SVVs, and parameters are created for that device.

Before a device may be used as the object of some action (e. g., before a *joint* may be *moved*) it must be *enabled*. Enabling a device is equivalent to starting the verb instance which controls the device.

Verb Composition

Verbs may be *composed*, or combined to form new verbs. A composition of verbs can be formally defined by taking unions of all the verb components; the verbs may then be sequenced by supplying an augmented transition mapping. In this way verbs with a highly specialized semantics can be assembled out of simpler, more general ones. (A verb with only one command list is called *simple*.) An example of this is "centering grasp" found in [8] and [9].

Levels of Control

The GPAC methodology supports programming for various *levels of control*: [9]

- *Closed-loop control* is achieved through low-level real-time computations such as application subroutines and data flow graphs.
- *Concurrency control* is concerned with specification of computations executing in parallel and is achieved through configuration of command lists and simple verbs.
- *Sequential control* deals with plans and strategies for executing complex motions or tasks and is achieved primarily by verb composition.

Programming for different levels of control involves different issues and areas of expertise; this is reflected in the methodology.

User Interface

AML/X

The GPAC user interface is a program running on the Programming System. Commands entered by a user are converted from a high-level form into actual messages to be sent to the Real-Time System. Also, configuration of system hardware and software is done via the user interface.

Because of the many abstractions and generic entities in the GPAC system, the high-level language AML/X was used to implement the user interface. AML/X is a language designed at IBM for automation programming and other applications [18]. It has a number of interesting features, among which the three most relevant to GPAC are:

- Data abstraction capability. An abstract data type is called a *class*; instances of classes consist of *instance variables* which can be manipulated using *methods*, a specialized type of subroutine call. The methods define the interface to the data type, except that some instance variables can be declared *exposed*, which makes them visible externally and hence part of the interface.
- Operator overloading. Operators can be defined on class instances, using a syntax similar to the method syntax. In particular, the act of applying parameters to an object is regarded as an operator in AML/X; this allows GPAC verb invocation to have the same syntactic form as subroutine invocation.
- Exception handling. AML/X has a rich set of constructs for raising and handling exceptional conditions [18].

AML/X can be easily interfaced to lower-level languages like C and FORTRAN. In fact, the communication between the PS and RTS in GPAC is handled by a C subroutine package which is called from AML/X.

Configuration and Execution Phases

The GPAC user interface consists of two parts: a *configuration phase* and an *execution phase*.

In the configuration phase the hardware layout is described, in terms of real-time processors available, interrupt sources, and physical devices attached to the processors. Next, previously compiled and linked object code modules are downloaded into the real-time processors. (For the most part, these modules are simply libraries of application subroutines linked with a real-time kernel.) Then generic objects such as function blocks, verbs, and logical device types are defined. Finally, some instances of logical devices may be created and enabled.

The execution phase consists primarily of invocation of verbs. Verbs are applied to devices to create verb instances, and commands concerning these verb instances are then passed to the real-time system.

Configuration and execution phases can be interleaved to a certain extent. New function blocks, verbs, and logical devices may be defined at any time, and existing verbs may be redefined. This can be done *while* real-time applications are running (as long as no instances of the affected verbs are still executing). The goal is to provide a highly interactive programming environment in which real-time system behavior can be modified in a very flexible and dynamic way. This meets the needs of both robotics researchers, who wish to experiment with alternative control strategies and new sensor and actuator technologies, and manufacturing engineers, who are responsible for reconfiguring workcells to meet changing work requirements.

Modes of Verb Invocation

The full life cycle of a verb instance can be described as follows:

- A verb is invoked by applying it to a parameter list. These parameters are processed by the PS and the information needed to create an instance of the verb is sent to the RTS.
- The RTS allocates and fills in the necessary data structures, and reports completion of this procedure to the PS.
- The PS sends a command to start the verb instance.
- The initial command list of the verb is executed on the RTS. The verb instance continues to execute until some condition forces it to terminate or until a *suspend* or *halt* message is received from the PS.
- When the verb instance terminates, the RTS sends notification to the PS. Depending on the reason for the termination, an exception may be raised on the PS.
- Finally, the PS sends a command to delete the verb instance. The RTS complies, cleaning up and deallocating all data structures.

In the simplest mode of invocation, this entire scenario is carried out synchronously. The user simply applies a list of arguments to a object of the verb class, e. g.:

```
move(joint,goal,speed);
```

The above operation will not complete until the corresponding verb instance terminates in the RTS and is deleted.

A more efficient, although more verbose, mode of invocation is provided by introducing an asynchronous phase:

```
vi: BIND move.asynch(joint,goal,speed);
/* other work can be done here */
vi.wait();
```

asynch is a verb method that results in creating and starting a verb instance, and returning a verb instance object without waiting for termination. At some later point in time, the verb instance can be waited for, and it is deleted after its termination.

Finally, a verb instance can be created and then *multiply invoked* with new input parameters each time:

```
vi: BIND move.new_instance(joint);
vi.start(goal,speed);
/* do something else */
vi.wait();
vi.start(another_goal,another_speed);
/* do something else */
vi.wait();
vi.delete();
```

Current Status and Future Work

The GPAC methodology has been implemented in conjunction with several different hardware architectures. The relative ease with which applications can be ported back and forth between these architectures is one encouraging result of our work. We have also observed that real-time computations are easy to program, require a minimal amount of debugging, and have predictable behavior once debugged. Finally, the system can be easily reconfigured: new devices and processors can be added without laborious changes and recompilations.

We have used a certain amount of formalism to describe the concepts underlying the methodology rather than relying on configuration examples in AML/X. We chose to do this because it is the formalism (not the current configuration syntax) which is really critical to understanding the methodology, because we did not wish to require detailed knowledge of AML/X of the reader, and because the syntax itself is subject to considerable change and enhancement as we gain experience with the system.

In the future, more sophisticated applications will be attempted using this methodology. We will then be able to judge the efficacy of the methodology for practical problems in automation. Graphical tools will be added to simplify verb and data flow graph configuration. Knowledge-based enhancements and natural-language-like interfaces are also contemplated.

References

1. F. Ozguner and M. L. Kao, "A Reconfigurable Multiprocessor Architecture for Reliable Control of Robotic Systems", *IEEE International Conference on Robotics and Automation*, St. Louis, March 1985.

2. S. Ahmad, "Real-Time Multi-Processor Based Robot Control", *IEEE Int. Conf. on Robotics and Automation*, San Francisco, April 1986.

3. V. Dupourque, H. Guiot, O. Ishacian, "Towards Multi-Processor and Multi-Robot Controllers", *IEEE Int. Conf. on Robotics and Automation*, San Francisco, April 1986.

4. I. Lee and S. Goldwasser, "A Distributed Testbed for Active Sensory Processing", *IEEE Int. Conf. on Robotics and Automation*, St. Louis, March 1985.

5. D. Siegel, et. al., "Computational Architecture for the Utah/MIT Hand", *IEEE Int. Conf. on Robotics and Automation*, St. Louis, March 1985.

6. L. S. Haynes and A. J. Wavering, "Real Time Control System Software: Some Problems and an Approach", *IEEE Int. Conf. on Robotics and Automation*, San Francisco, April 1986.

7. R. D. Gaglianello, "A Distributed Conputing Environment for Robotics", *IEEE Int. Conf. on Robotics and Automation*, San Francisco, April 1986.

8. J. U. Korein, G. E. Maier, R. H. Taylor, and L. F. Durfee, "A Configurable System for Automation Programming and Control", *IEEE International Conference on Robotics and Automation*, San Francisco, April 1986.

9. G. E. Maier, R. H. Taylor, J. U. Korein, "A Dynamically Configurable General Purpose Automation Controller", *Fourth IFAC/IFIP Symp. on Software for Computer Control*, Graz, Austria, May 1986.

10. S. N. Griffiths, "Design Methodologies - A Comparison", in *Tutorial: Software Design Strategies*, G. Bergland and R. Gordon, eds. 1979. pp. 189-213.

11. G. D. Bergland, "Structured Design Methodologies", in *Tutorial: Software Design Strategies*. pp. 162-181.

12. U. S. Dept. of Defense, *Reference Manual for the Ada Programming Language*.

13. V. Dupourque, "Using Abstraction Mechanisms to Solve Complex Task Programming in Robotics", *IEEE Int. Conf. on Robotics and Automation*, San Francisco, April 1986.

14. M. D. Donner, "The Design of OWL: A Language for Walking", *Proceedings of the SIGPLAN '83 Symp. on Prog. Lang. Issues in Software Systems*. pp. 158-165

15. K. G. Shin and M. E. Epstein, "Communication Primitives for a Distributed Multi-Robot System", *IEEE Int. Conf. on Robotics and Automation*, St. Louis, 1985.

16. K. Schwan, T. Bihari, B. W. Weide, and G. Taulbee, "GEM: Operating System Primitives for Robots and Real-Time Control Systems", *IEEE Int. Conf. on Robotics and Automation*, St. Louis, 1985.

17. B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, 1978. July 1982.

18. *IBM Manufacturing System: A Manufacturing Language Reference Manual*, No. 8509015, IBM Corporation, 1983.

19. L. R. N[illegible]man and R. H. Taylor, "A Hierarchical Exception Handler Binding Mechanism", *Software - Practice and Experience*, 14, 10, Oct. 1984. pp. 999-1007.